CHAPTER OBJECTIVE

- Context free grammar and Context Free Language
- Deterministic Context Free Language (DCFL)
- Derivations and derivation (Parse) trees
- Sentential forms
- Right most and left most derivation of strings
- Ambiguity in grammar and language
- Useless symbols in CFG
- Construction of Reduced CFG
- Elimination of null and unit productions
- Chomsky and Greibach normal form
- CYK algorithm
- Applications of CFG

Context-free grammars (CFGs)

- Context-free grammars were first used in the study of human languages.
- The context-free grammars are used as basis in the design and implementation of compiler.
- The designers of compilers often use such grammars to implement components of compiler, like parsers, scanners, code generators, etc.
- Any implementation of programming language is preceded by a context-free grammar that specifies it.
- Important applications of context-free grammar theory have been made to compiler design.
- The validity of high level language statements is checked by context free grammar and context free language by constructing derivation trees or parse trees.
- Context-free languages are applied in parser design and Chomsky normal form is used in decidability (emptiness of CFG) concept of push down automata.

DEFINITION OF CONTEXT FREE GRAMMAR

A context-free grammar (CFG) is defined as 4-tuples (V_N, Σ, P, S), where P is a set of production rules of the form

one nonterminal → finite string of terminals and/or nonterminals

This general form of productions in CFG, states that the nonterminal in left hand side of the production is free from any context (there is neither left nor right context), therefore these type of grammars are called context-free grammars.

***G* is a context-free grammar if every production it has is of the form.**

$A \rightarrow \alpha$

where, $A \in V_N$, $\alpha \in (V_N \cup \Sigma)^*$

Good to know

Production rules are also known as grammar rules. Some text authors use the symbol '::=' instead of '→' in productions. Also, some authors indicate nonterminals (variables) by writing them within angle brackets like

<S> → <X> |<Y> | <Z>

<Z> → a |a<X>

<Y > → b|b<Y>

<Z > → ^

Here S, X, Y, Z ∈ V_N, and a, b ∈ Σ

Good to know

(*i*) *A grammar G, defined as* $E \rightarrow E + E, E \rightarrow E*E, E \rightarrow (E), E \rightarrow id$, *can also be written as* $E \rightarrow E + E \mid E * E \mid (E) \mid id$. *We will use both formats in the book.*

(*ii*) *When we deal more than one grammars simultaneously, we will use notation* G_1 *or* G_2 *or something else with derivation symbol* $\Rightarrow$ *to make sure a particular derivation corresponds to which grammar. For example,* $S \overset{*}{\Rightarrow}_{G_k} w$ *states that the string 'w' is derived from production rules of grammar* G_k *in more than one steps.*

CONTEXT FREE LANGUAGE

- **The languages generated by the context-free grammars are called context free languages.**
- **An important example for context-free languages is the syntax of programming languages.**
- **Context-free languages are also applied in the design of parser.**
- **These are also useful for describing block structure in programming languages.**
- **Context free languages are more complicated than regular languages.**

Deterministic Context Free Language (DCFL)

- **A language *L* is a deterministic context free language (abbreviated as DCFL) if there is a deterministic push down automaton (abbreviated as DPDA) accepting language *L*.**
- **A push down automaton *M* is said to be deterministic if there is no configuration for which *M* has a choice of more than one move in same input conditions.**
- **A context-free grammar is deterministic if and only if its rules allow a (left-to-right, bottom-up) parser to proceed without having to guess the correct continuation at any point.**
- **A language is deterministic context-free if and only if it is generated by some deterministic context-free grammar.**
- **The language being described is in fact nondeterministic; it is impossible to write a deterministic context-free grammar for it.**

DERIVATIONS

- A derivation can be defined as the process to generate string of terminals by replacing the nonterminals in right hand side of a production.
- If the terminal string '*s*' is generated from an *S*-production of grammar *G* then $s \in L(G)$.
- The symbol ⇒ is used to represent a derivation.
- To distinguish whether it is a left most derivation or a right most derivation we use notation LMD and RMD with ⇒. This way the notations $\overset{*}{\underset{lm}{\Rightarrow}}$ and $\overset{*}{\underset{rm}{\Rightarrow}}$ represent leftmost and rightmost derivations respectively.
- The derivation is the process of generation of terminal string from a given set of productions.
- The yield of a derivation tree is also known as derivations. If *s* = '*aba*' is a derivation, it means '*s*' is the string obtained by concatenating the labels of leaf nodes from left to right.

PARSE (DERIVATION) TREES

The strings generated by a context free grammar $G = (V_N, \Sigma, P, S)$ can be represented by a hierarchical structure called tree.

A parse tree (also called derivation tree, syntax tree, production tree, generation tree) for a context free grammar G has following characteristics:

i. **Every vertex of parse tree has a label which is either a variable (generally denoted by an uppercase letter and also called nonterminal) or a terminal from $\{\Sigma \cup \wedge\}$.**
ii. **The root of parse tree has label S, where S is start symbol.**
iii. **The label of an internal vertex is always a variable or nonterminal.**
iv. **If a vertex A has k children with labels $A_1, A_2, A_3, A_4, \ldots\ldots, A_k$, then there exists a production $A \rightarrow A_1A_2A_3A_4\ldots\ldots A_k$ in context free grammar G.**
v. **A vertex is an intermediate node if its label is $N \in V_N$.**
vi. **A vertex is a leaf node if its label is $n \in \{\Sigma \cup \wedge\}$.**
vii. **A vertex is the only child of its parent node if its label is $\wedge$.**

The nonterminal *A* can be replaced by '*ba*' if there exists a production

$A \rightarrow ba$

and following subtree can be drawn for this production

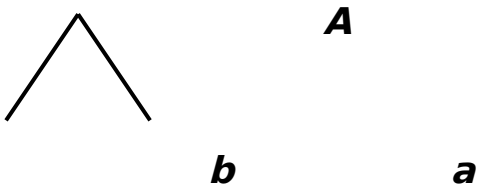

Fig. 6.4 A derivation tree for the production $A \rightarrow ba$

Definitions

- **Yield of a derivation tree.** The yield of a derivation tree is the concatenation of the labels of leaf nodes in left-to-right ordering without repetition. For example the yield of derivation tree given by figure 6.5 (a) is '*bba*'
- **Subtree of a derivation tree.** A subtree of a derivation tree is a tree having following characteristics:
  - ✓ The root of a subtree is some vertex '*v*' of derivation tree, where $v \in V_N$.
  - ✓ The vertices of subtree are descendents of '*v*', with their labels, and
  - ✓ The edges of subtree connect the descendents of '*v*'.

Definitions

- **A-tree.** A subtree looks like a derivation tree except that the label of the root node may not be S (the start symbol) if and only if S does not appear in the right side of any production. It is called an *A-tree* if the label of its root is A. similarly if there is another subtree whose label of root node is B then it is called a *B-tree*.
- **Working String.** The arrow symbol '⇒' is employed between the unfinished stages of the generation of a string or a derivation. It means that there exists a possible substitution in $XY \Rightarrow abY$ if there exists a production $X \rightarrow ab$. These unfinished stages are strings of terminals and nonterminals that are generally called working strings.
- **Internal Nodes.** A node in a derivation tree whose label is a variable (or say nonterminal) is called a leaf node or an external node. For example, if the label of a node is $A \in V_N$, then A is an internal node.
- **Leaf Node.** These are also called external nodes. A node in a derivation tree, whose label is a terminal $a \in \Sigma$ or $\wedge$ is called a leaf node or external node or simply a leaf.
- **Sentential Form.** A string of terminals and variables (say α) is called a sentential form if $S\alpha$, where $S \in V_N$ and $\alpha \in (V_N \cup \Sigma)^*$.
- **Nullable Variable.** A variable E in CFG G is said to be nullable if it derives (or say generates) $\wedge$ *i.e.*, $E \Rightarrow \wedge$.

FROM INFERENCE TO TREE

Theorem 6.1 If $G = (VN, \Sigma, P, S)$ is a context-tree grammar, then $S \Rightarrow \alpha$ if and only if there is a derivation tree in grammar G with yield α.

PROOF. First of all we would like to prove that $A \Rightarrow \alpha$ if and only if there is A-tree with yield α. Once this is proved, theorem will also be proved by assuming that $A = S$.

We prove that $A \Rightarrow \alpha$ by induction on the number of internal vertices in an A-tree (say T). When the tree T has only one internal vertex, the remaining vertices are leaf nodes and are children nodes of root A, as given by figure 6.6 below :

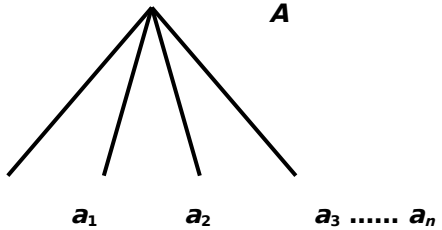

Fig. 6.6 A derivation tree for $A \Rightarrow a_1\ a_2\ a_3 \ldots\ldots\ a_n$

By definition of derivation tree, derivation tree given by figure 6.6, shows that there must be a production

$A \rightarrow a_1\ a_2\ a_3 \ldots\ldots\ a_n$ (where $a_1\ a_2\ a_3 \ldots\ldots\ a_n = \alpha$)

or we can say that there is a derivation

$A \Rightarrow \alpha$

Thus there is a basis for induction. Let us assume the result for all trees with at most *k* – 1 internal nodes (*i.e.*, the nodes whose labels are variables), where *k* > 1.

Let us assume that *T* is an *A*-tree with *k* internal nodes, where *k* ≥ 2. Let *p*1, *p*2,..... , *Pn* be the children of the root in the left-to-right ordering, with their labels *X*1, *X*2, *X*3,.....*Xn* respectively. Therefore, there must be a production

A* → *X*1 *X*2 *X*3.....*Xn

in *P*, and there is a derivation *A* *X*1 *X*2 *X*3.....*Xn*

As *k* ≥ 2, then there is at least one child which is internal node. According to left-to-right ordering of leaves, α can be written as α1α2α3..... α*n*, where each α*i* is obtained by the concatenation of the labels of the leaves which are descendents of vertex *pi*. If *pi* is an internal vertex, consider the subtree *T* with *pi* as its root. The number of internal nodes of the subtree is less than *k* (as there are *k* internal nodes in *T* and at least one of them, *i.e.*, its root is not in the subtree). So by the induction hypothesis applying to the subtree,

$$X_i \overset{*}{\Rightarrow} \alpha_i$$

If *pi* is not an internal node (*i.e.*, it is a leaf node), then *Xi* = α*i*. By using the production A→*X*1*X*2*X*3....*Xn*, we get the derivation

***A* ⇒ *X*1 *X*2 *X*3....*Xn*,**

⇒ *X*2 *X*3....*Xn*,

.

.

⇒ α1α2α3..... α*n*

= α

or we can say $A \overset{*}{\Rightarrow} \alpha$. According to the principle of induction, $A \overset{*}{\Rightarrow} \alpha$ whenever α is the yield of an *A*-tree. For the proof of 'only if' part, let us assume that $A \overset{*}{\Rightarrow} \alpha$. We have to construct an *A*-tree whose yield is α. We can do this induction through the number of steps to get $A \overset{*}{\Rightarrow} \alpha$. If there is a derivation $A \Rightarrow \alpha$, then $A \rightarrow \alpha$ is a production in *P*. If $\alpha = X1\ X2\ X3....Xn$, , the *A*-tree with yield is constructed, as given by figure 6.7.

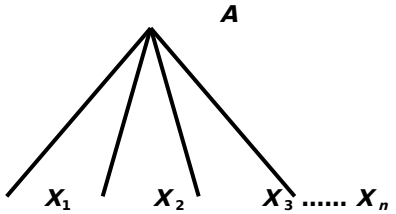

Fig. 6.7 A derivation tree for one step derivation

This is the basis for induction. Suppose the result for derivations takes at most *k* steps. Suppose $A \overset{k}{\Rightarrow} \alpha$

We can split this derivation as

$$A \Rightarrow X1X2\ X3.....Xn \quad \alpha$$

$$\overset{k-1}{\Rightarrow}$$

where $A \Rightarrow X1\ X2\ X3.....Xn$ implies the production $A \rightarrow X_1\ X_2\ X_3.....X_n$ in P. In the derivation $X1X2\ X3.....Xn \overset{k-1}{\Rightarrow} \alpha$, there is one case out of two cases :

(i) X_i is not changed throughout the derivation, or

(ii) X_i is changed in some subsequent step

Let α_i be the substring of derived from X_i. Then

in case (i) $X_i \overset{*}{\Rightarrow} \alpha_i$

and

in case (ii) $X_i \Rightarrow \alpha_i$

As G is context-free grammar, therefore in every step of the derivation $X_1X_2\ X_3.....X_n \Rightarrow \alpha$

We replace a single variable by a string. As $\alpha_1, \alpha_2, ... \alpha_m$ account for all the symbols in α.

$\alpha = \alpha_1\alpha_2\alpha_3.....\alpha_n$.

We construct the derivation tree with yield α, as follows :

As $A \rightarrow X_1\ X_2\ X_3.....X_n \in P$, we construct a derivation tree with n external (leaf) nodes whose labels are $X_1X_2X_3.....X_n$ in the left-to-right ordering. The constructed derivation tree is given by Figure 6.8.

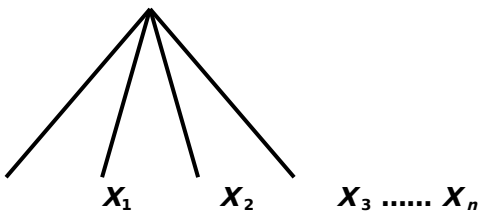

Fig. 6.8 A derivation tree with yield $X_1X_2X_3.....X_n$

In case (i), we leave the node p_i as it is. In case (ii), in less than k steps (as $X1X2X3....Xn\ \alpha$. According to induction hypothesis three exists an Xi -tree Ti with yield Ti. We attach the tree Ti at the node p_i, as p_i is root of tree Ti. The resulting constructed tree is given by Figure 6.9.

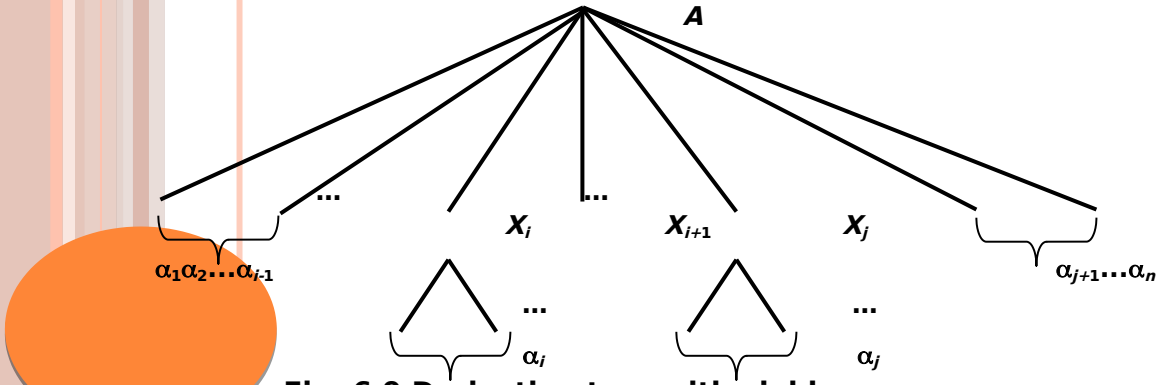

Fig. 6.9 Derivation tree with yield $\alpha_1\alpha_2\alpha_3.....\alpha_n$

In the Figure 6.9, let use suppose i and j are the first and last indexes such that X_i and X_j satisfy (ii) case. Therefore, $\alpha_1\alpha_2...\alpha_{i-1}$ are the labels of leaves at level 1 in T. α_i is the yield of the X_i-tree T_i, α_{i+1} is the yield of the X_{i+1}-tree T_{i+1}, and so on.

Thus we get a derivation tree with yield α. By the principle of induction we get the result for any derivation. This completes the proof of 'only if' part of the theorem.

DERIVATION TREE AND NEW NOTATION OF ARITHMETIC EXPRESSIONS

In the special type of context-free grammar of operators and operands, we can construct meaningful derivation trees that enable us to introduce a new notation for arithmetic expression which has direct applications to computer science.

These notations are prefix and post fix notations. Compilers often convert infix to prefix notation and then to assembler code.

From a derivation tree of an algebraic expression, we can get equivalent prefix and postfix notations. An algebraic expression in terms of operators and operands can be derived by an ambiguous context-free grammar.

Prefix notation is the parenthesis-free notational scheme invented by Polish logician Jan Lukasiewicz (1878-1966) and is often called polish notation.

In prefix notation operator is followed by operands. For example, in prefix notation *A*+*B* is written as +*AB*. Postfix notation is reverse of prefix notation *AB*+ is the equivalent postfix notation of *A*+*B*.

First of all the derivation tree of expression (infix form) generated by context-free grammar, is constructed. Let us suppose the given ambiguous CFG *G* is defined by following productions

$S \rightarrow (S)$

$S \rightarrow S + S$

$S \rightarrow S - S$

$S \rightarrow S * S$

$S \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Here we are assuming that every operated is of single digit. Suppose the generated algebraic expression by grammar *G* is

$S \Rightarrow ((6 + 5)* (3 - 4) + 2)*1$

We can easily construct the derivation tree for this expression by using the production sequences we have applied. Note that there is no need to show parentheses in derivation tree. The constructed derivation tree for ((6 + 5)* (3 – 4) + 2)*1 is given by figure 6.10.

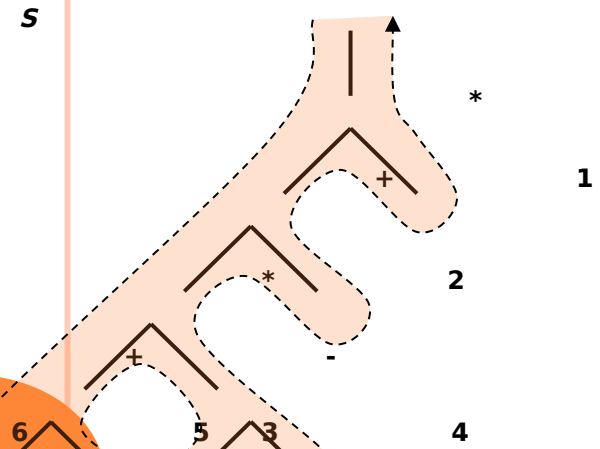

Fig. 6.10 Derivation tree for expression ((6 + 5)* (3 – 4) + 2)*1 with direction for evaluation of prefix notations

As the derivation tree is unambiguous, the prefix notation is also unambiguous. To convert the infix expression ((6 + 5)* (3 – 4) + 2)*1 into equivalent prefix notation we need a binary tree (here we consider derivation tree given by figure 6.10) in the sequence specified by arrows. Scanning this way, we get

***, +, *, 6, 5, –, 3, 4, 2, 1**

This is the equivalent prefix expression. Make sure that neither an operator nor an operand is scanned more than once. Here we are not concerned with how this prefix expression will be evaluated.

SENTENTIAL FORMS

- The set of all terminal strings represented by start symbol constitute the language of a CFG. Let $G = (V_N, \Sigma, P, S)$ be a CFG and there is some string $s \in (V_N, \Sigma,)^*$ such that $S \overset{*}{\Rightarrow} s$, then there exists a sentential form.
- If the sentential form is obtained through a leftmost derivation then it is called left sentential form.
- Similarly if the sentential form is obtained through a rightmost derivation then it is called right sentential form.
- If the number of terminal strings represented by the start symbol are finite then there will be finite number of derivation or parse trees to represent those terminal strings otherwise there will be infinite number of derivation trees.

Definitions

Leftmost Derivation. A derivation $S \overset{i}{\Rightarrow} s$ is said to be a leftmost derivation if production is applied only to the left most variable (nonterminal) at every step. It is not necessary that we can derive a derivation by start symbol only; we can also obtain a derivation from some other variable in V_N.

Let us consider a CFG *G* having productions *S* → *S* - *S* | *S* * *S* | (*S*) | *a*. For clarity, the leftmost variable in derivation is in bold style. Suppose, first of all we use production *S* → *S* - *S* *S* ⇒ *S* – *S*. Now we can apply a production on left most variable. We have

S* ⇒ *S* * *S* - *S

Now we apply as S-production (here *S* → *a*) on left most *S* to get

S* ⇒ *a* * *S* – *S

⇒ a * a – S

⇒ a * a - a

In other words, we can say that

$S \overset{*}{\Rightarrow} a * a - a$

where, *a* * *a* – *a* is the left most derivation. For more precise description we write

$S \overset{*}{\underset{LM}{\Rightarrow}} a * a - a$

Definition

Rightmost Derivation. A derivation $S \overset{*}{\Rightarrow} s$ is said to be a rightmost derivation if production is applied only to the right most variable (nonterminal) at every step. It is not necessary that we can derive a derivation by start symbol only; we can also obtain a derivation from some other variable in V.

Let us again consider a CFG *G* having productions $S \rightarrow S - S \mid S * S \mid (S) \mid a$. Now we will find a right most derivation by using this grammar. During derivation steps, we will represent rightmost variable in bold style. Let us start with the production $S \rightarrow S - S$

$S \Rightarrow S - S$

$\Rightarrow S - a$

$\Rightarrow S * S - a$

$\Rightarrow S * a - a$

$\Rightarrow a * a - a$

In other words, we can say that

$$S \overset{*}{\Rightarrow} a * a - a$$

where, $a * a - a$ is the right most derivation. For more precise description we write

$$S \overset{*}{\underset{RM}{\Rightarrow}} a * a - a$$

Here, *RM* denotes the right most derivation.

Let us consider the context free grammar G having two production $S \rightarrow S + S \mid a$. The string $a + a + a$ has two distinct left most derivations :

$$S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + S + S \Rightarrow a + a + S \Rightarrow a + a + a$$

And

$$S \Rightarrow S + S \Rightarrow S + S + S \Rightarrow a + S + S \Rightarrow a + a + S \Rightarrow a + a + a$$

The corresponding derivation trees are given by figure 6.11(a), (b).

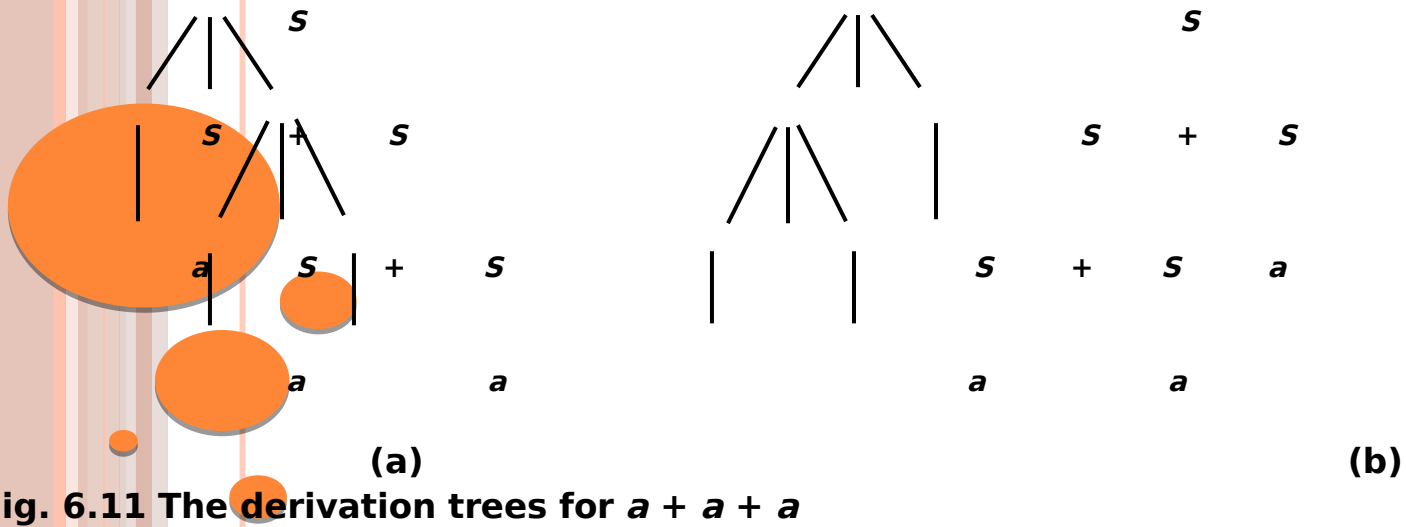

Fig. 6.11 The derivation trees for $a + a + a$

Definition

Total Language Tree. For a given context-free grammar *G*, we define a derivation tree with the start symbol as it root node, and its children as working string of terminals and nonterminals. The descendents of each node are all possible results of applying every applicable production to the working string, one at a time. A string of all terminals is a terminal node in the tree. This type of resultant tree is called the total language tree of the context free grammar *G*.

For example, if a CFG *G* consists of following productions :

***S* → *aa* | *bA*| *aAA*, *A* → *b* | *a*.**

The total language tree for this CFG is given by figure 6.12 (see below).

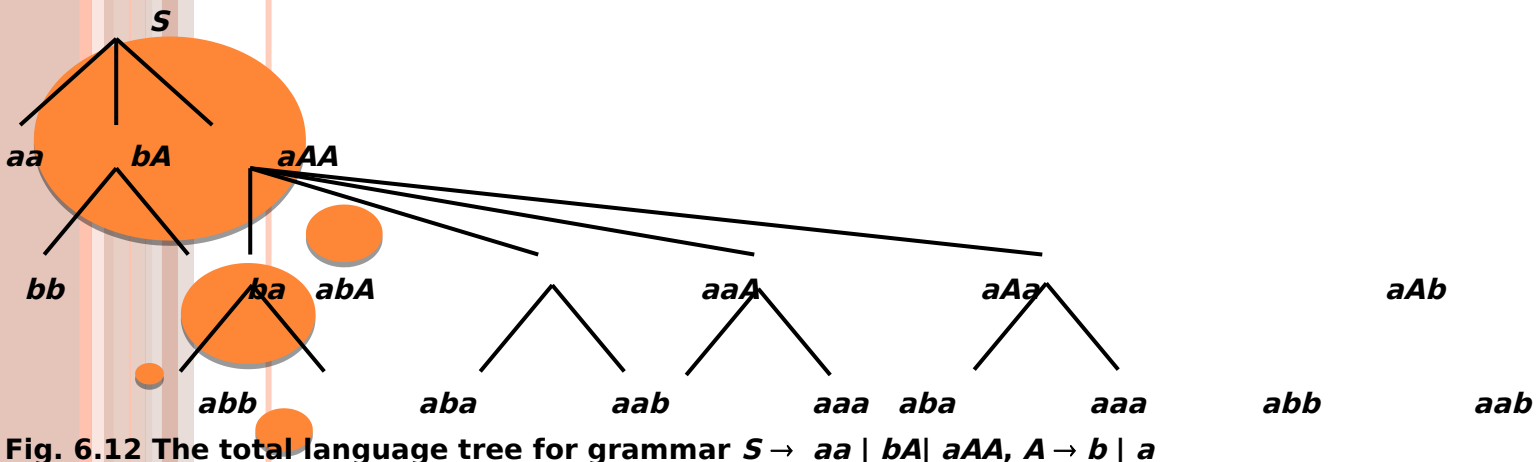

Fig. 6.12 The total language tree for grammar *S* → *aa* | *bA*| *aAA*, *A* → *b* | *a*

Example 6.5 Consider the context-free G, that consists of following production :

$S \rightarrow aB \mid bA,\ A \rightarrow a \mid aS \mid bAA,\ B \rightarrow b \mid bS \mid aBB$

For the string *aabbabab*, find (a) leftmost derivation (b) right most derivation (*c*) parse tree.

SOLUTION. (a) As the string *aabbabab* has first symbol *a*, therefore first of all we consider the production $S \rightarrow aB$ for leftmost derivation. Following are the leftmost derivation steps :

$S \Rightarrow aB$

$\Rightarrow aaBB$ (After applying $B \rightarrow aBB$)

$\Rightarrow aabB$ (After applying $B \rightarrow b$)

$\Rightarrow aabbS$ (After applying $B \rightarrow bS$)

$\Rightarrow aabbaB$ (After applying $S \rightarrow aB$)

$\Rightarrow aabbabS$ (After applying $B \rightarrow bS$)

$\Rightarrow aabbabaB$ (After applying $S \rightarrow aB$)

$\Rightarrow aabbabab$ (After applying $B \rightarrow b$)

(b) For rightmost derivation again we will consider the production $S \rightarrow aB$. In this derivation we will apply production on rightmost variable at every step. Following are the derivation steps :

$S \Rightarrow aB$

$\Rightarrow aaBB$ (After applying $B \rightarrow aBB$)

$\Rightarrow aaBbS$ (After applying $B \rightarrow bS$)

$\Rightarrow aaBbaB$ (After applying $S \rightarrow aB$)

$\Rightarrow aaBbabS$ (After applying $S \rightarrow bS$)

$\Rightarrow aaBbabaB$ (After applying $S \rightarrow aB$)

$\Rightarrow aaBbabab$ (After applying $B \rightarrow b$)

$\Rightarrow aabbabab$ (After applying $B \rightarrow b$)

(c) The parse tree for string *aabbabab* is given by figure below (figure 6.22) :

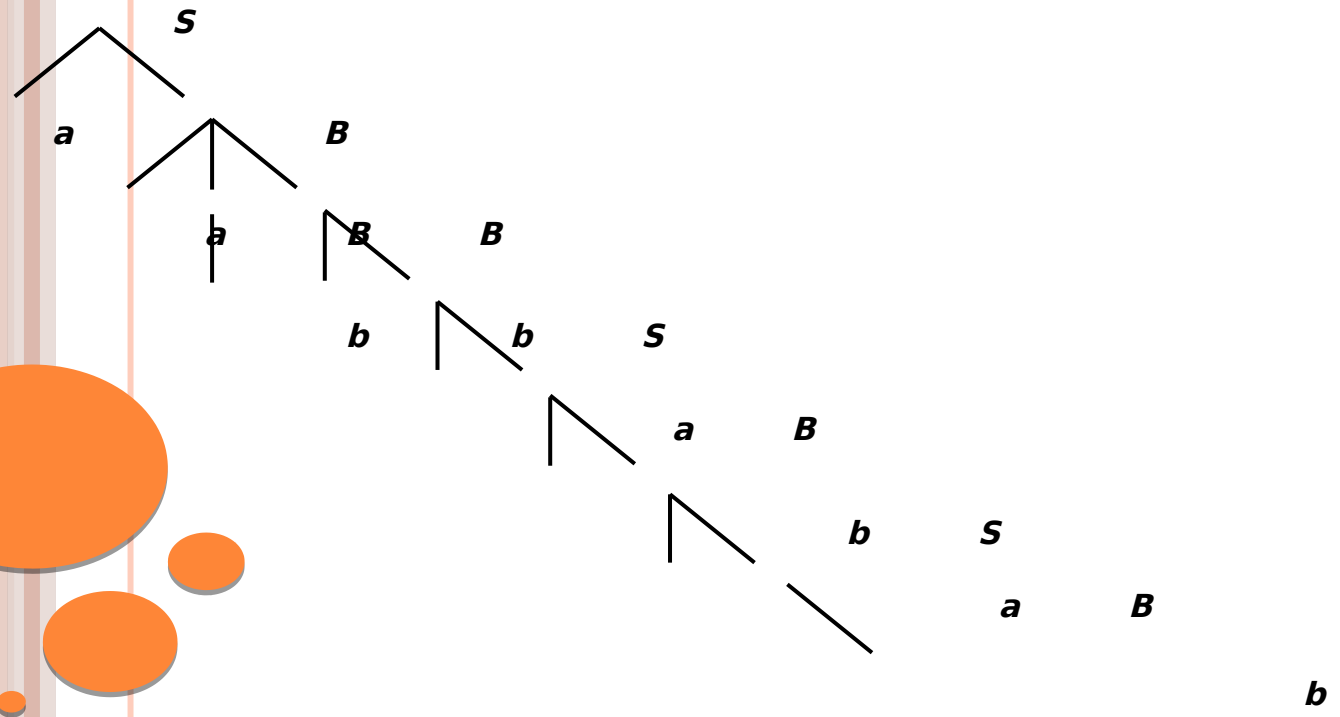

Fig. 6.22 Derivation tree for leftmost derivation of example 6.5

Similarly, the derivation tree for rightmost derivation can also be drawn.

Definition

Ambiguous Grammar. A context free grammar G is said to be ambiguous if there exists at least one string $s \in L(G)$ having two or more distinct derivation trees (or equivalently, two or more distinct left most derivations; or two or more distinct right most derivations).

Example 6.6 Show that the language of all non-null strings of *a*'s defined by a context-free grammar $G = (\{S\}, \{a\}, \{S \rightarrow aS \mid Sa \mid a\}, S)$ is ambiguous.

SOLUTION. By using the grammar *G*, the string *a*3 (or say *aaa*) can be derived by four different ways given by figure 6.23 (a) – (d).

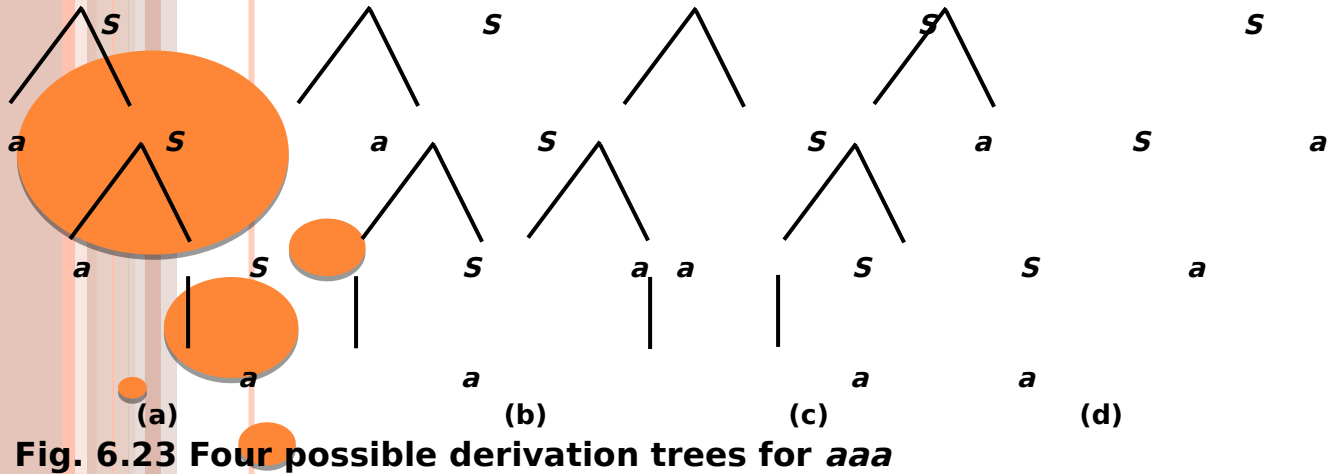

Fig. 6.23 Four possible derivation trees for *aaa*

The derivation steps for (a) are :

$S \Rightarrow aS$

$\Rightarrow aaS$

$\Rightarrow aaa$

The derivation steps for (b) are :

$S \Rightarrow aS$

$\Rightarrow aSa$

$\Rightarrow aaa$

The derivation steps for (c) are :

$S \Rightarrow Sa$

$\Rightarrow aSa$

$\Rightarrow aaa$

The derivation steps for (d) are :

$S \Rightarrow Sa$

$\Rightarrow Saa$

$\Rightarrow aaa$

A set of four derivation trees for string a^3 (*i.e.*, *aaa*) shows that grammar *G* is ambiguous.

Definition

Inherently Ambiguous CFL. A context-free language for which every context-free grammar is ambiguous is said to be an inherently ambiguous context-free language. For example, the language

$$L = \{a^n b^n c^m d^m \mid m, n \geq 1\} \cup \{a^n b^m c^m d^n \mid m, n \geq 1\}$$

is inherently ambiguous because there are infinitely many strings of the form $a^n b^n c^n d^n$, for $n > 1$, which must have two distinct left most derivations.

Let us consider the context free grammar *L*(*G*) with following productions: *S* → *A* | *BC*

A* → *aAd* | *aDd

D* → *bDc* | *bc

B* → *aBb* | *ab

C* → *cCd* | *cd

This grammar with start symbol S generates the language

$$L(G) = \{a^n b^n c^m d^m \mid m, n \geq 1\} \cup \{a^n b^m c^m d^n \mid m, n \geq 1\}$$

For string *aabbccdd* ∈ *L*(*G*), we have two following different left most derivations:

- ***S* ⇒ *A* ⇒ *aAd* ⇒ *aaDdd* ⇒ *aabDcdd* ⇒ *aabbccdd***
- ***S* ⇒ *BC* ⇒ *aBbC* ⇒ *aabbC* ⇒ *aabb cCd* ⇒ *aabbccdd***

AMBIGUOUS TO UNAMBIGUOUS CONTEXT FREE GRAMMAR

- **Some context free languages are inherently ambiguous in the sense that they cannot be generated except by ambiguous grammars.**
- **Ambiguity is usually a property of grammar rather that the language.**
- **If a context free grammar is ambiguous, it is often possible and desirable to find an equivalent unambiguous grammar context free grammar.**

Let us consider a context free grammar $G = (V_N, \Sigma, P, S)$, with operators +, *, (,), and *id* in Σ. Suppose, P contains the production

$S \rightarrow S + S$

$S \rightarrow S * S$

$S \rightarrow (S)$

$S \rightarrow id$

To determine an equivalent unambiguous context free grammar with other operators is slightly different.

- As we see the production $S \rightarrow S + S$ is itself enough to produce ambiguity. Therefore, we need to eliminate the productions of this form.
- As we want to eliminate the productions of the form $S \rightarrow S + S$, so we will not think about expressions involving + as being sum of other expressions.
- Let us use the letter T to stand for the things which are added to create expressions, where T is nonterminal.
- Like sums, the expressions can also be product. However two expressions $a + b * c$ and $a * b + c$ are both sums, but it is more appropriate to say that the term T can be product.
- Additionally, let us say that 'factors' are the things that are multiplied to produce terms. Let us represent these terms by nonterminal F.

At this stage, expressions are sums of one or more terms, terms are products of one or more factors, and factors are either expression inside parentheses or single identifier. Hence, the 'sum of terms' can be represented as

$S \rightarrow T + T \mid T$

But again we would have ambiguity in the grammar. What we say instead is that an expression is either a single term or the sum of a term with some other expression. The only point is whether we want

$S \rightarrow S + T$ or $S \rightarrow T + S$

If we wish to represent $a + b + c$ as $(a + b) + c$, we would probably choose $S \rightarrow S + T$ as more appropriate. In other words, an expression with more than one term is obtained by adding the last term to the sub expression containing all but the last term. For same reason we choose, the production

$T \rightarrow T * F$

rather than the production $T \rightarrow F * T$. The resultant unambiguous context free grammar is

$G = (V_N, \Sigma, P', S)$

where,

$V_N = \{S, T, F\}$

$P' = \{ S \rightarrow S + T \mid T,\ T \rightarrow T * F \mid F,\ F \rightarrow (S) \mid id \}$

We must now need to prove two things :

G' is equivalent to G (given ambiguous CFG)

G' is the unambiguous CFG

USELESS SYMBOLS IN CFG

- There are several ways in which we can restrict the format of production rules in a context-free grammar without reducing the capabilities (power to generate language) of grammar.
- In a context-free grammar *G*, it may not be necessary to use all the variables and terminals, or all the productions in *P* for generating the strings.
- We always try to eliminate productions, terminals, and nonterminals in *G*, which are not useful for derivation of strings belonging to *L*(*G*).

Let $G = (V_N, \Sigma, P, S)$ be a context-free grammar. A symbol '*A*' is useful if there is a derivation

$$S \overset{*}{\Rightarrow} \alpha A \beta \overset{*}{\Rightarrow} s$$

for some α, β and *s*, where $s \in \Sigma^*$. Otherwise '*A*' is useless. There are two aspects to usefulness. First some terminal string must be derivated from '*A*' and second '*A*' must occur in some string derivable from *S* (the start symbol).

If $L(G)$ is a nonempty context-free language (*i.e*, it generates at least one terminal string), then it can be generated by a CFG G with following properties :

Each terminal and nonterminal must be involved in the derivation of some string in $L(G)$.

There are no productions of the form $A \rightarrow B$ (called unit production) where $A, B \in VN$.

If $\wedge \notin L(G)$, then there is no need of productions of the form $A \rightarrow \wedge$. Let us consider a context-free grammar

$G = (\{S, A, B, C, D,\}, \{0, 1, 2\}, P, S)$,

where P is defined as :

$P = \{S \rightarrow AB, A \rightarrow 01, B \rightarrow C \mid 1, D \rightarrow 12 \mid \wedge\}$

We take a string $s = 011$, such that $011 \in L(G)$.

Let us assume another grammar $G = (\{S, A, B,\}, \{0, 1\}, P_1, S)$, where $P1$ is defined as :

$P_1 = \{S \rightarrow AB, A \rightarrow 01, B \rightarrow 1\}$.

We see that $L(G) = L(G1)$. It means the language generated by both grammars G and $G1$ are same. From grammar $G1$, we have eliminated production $B \rightarrow C$ and $D \rightarrow 12 \mid \wedge$ which are useless productions in grammar G, on the following basis :

i. The variable C does not derive any terminal string.

ii. D and 12 are not involved in any sentential form.

iii. $B \rightarrow C$ simply replace B by C (unit production).

iv. $D \rightarrow \wedge$ is a null production.

Therefore for the simplification of CFG, it is necessary to eliminate the productions, variable and terminals having following characteristics :

v. a production in which a variable does not derive a terminal string

vi. the symbols ($V_N \cup \Sigma$), not appearing in any sentential form

vii. unit productions (*e. g.*, $A \rightarrow B$)

viii. null productions (*e. g.*, $A \rightarrow \wedge$).

Construction of Reduced CFG

Theorem 6.4 If G is a context grammar that generates non-empty language $L(G)$ (*i.e.* $L(G) \neq \phi$), then we can find an equivalent reduced *CFG* G_1 such that each variable in G_1 derives some terminal string.

PROOF. Suppose $G = (V_N, \Sigma, P, S)$ be the given context-free grammar. Let us construct $G_1 = (V_N', \Sigma, P', S')$ as reduced context-free grammar by following steps :

Step 1. Construction of V_N'. We define $W_i \subseteq V_N$. It means that W_i is subset of V_N, which says that all element of W_i will definitely be in V_N. But the reverse must be true, it is not necessary. We define W_1 as

$$W_1 = \{A \in V_N \mid \text{there exists a production } A \to s \text{ where } s \in \Sigma^*\}$$

If $W_1 = \{\phi\}$, some variable will remain after the application of any production, and so $L(G) = \phi$.

Each W_{i+1} is constructed in terms of W_i as :

$$W_{i+1} = W_i \cup \{A \mid A \in V_N$$

$$\mid \text{there exists some production where}$$

According to definition of W_i , $Wi \subseteq W_{i+1}$ for all i. It means that all the variables in W_{i+1} may also be in W_i but definitely all the variables in W_i must be in W_{i+1}.

As V_N has only finite number of variables, then there will be a condition that

$$Wk = Wk+1 \text{ for some } k \leq |V_N|$$

Therefore, $Wk = Wk + m$ for $m \geq 1$. At this stage we define

$V_N' = Wk$

for newly constructed grammar G_1.

Step 2. Construction of P'. The set of production rules P' is defined as :

$$P' = \{A \rightarrow \alpha \mid A, \alpha \in (V_N' \cup \Sigma)^*\}$$

This means P' contains only those production that involve the variables in V_N'. Hence we can define the reduced grammar as

$G_1 = (V_N', \Sigma, P', S)$

where $S \in V_N'$. Note that the grammar G_1 is not completely reduce grammar because we have not proved that all the variables and terminals in G' are fully involved in deriving sentential form.

Theorem 6.5 For every context grammar $G_1 = (V_N', \Sigma, P, S)$, we can construct an equivalent grammar $G' = (V_N'', \Sigma'', P'', S)$ such that every symbol in $(V_N'' \cup \Sigma'')$ appears in some sentential form.

PROOF. The theorem states that for every $Z \in (V_N'' \cup \Sigma'')$ there exists β such that and Z is a symbol in the string β. Note that the Σ'' is the reduced Σ.

We construct reduced grammar $G' = (V_N'', \Sigma'', P'', S)$ by following steps :

Step 1. Construction of V_N''. We construct W_i for $i \geq 1$. First we define $W_1 = \{S\}$, then we determine W_{i+1} in terms of W_i for each $i \geq 1$. We define $Wi+1$ as :

$W_{i+1} = W_i \cup \{ Z \in (V_N' \cup \Sigma) \mid$ there exists a production $A \rightarrow \beta$ with $A \in W_i$ and β containing the symbol $Z \}$.

Note that $W_i \subseteq (V_N' \cup \Sigma)$ and $W_i \subseteq W_{i+1}$. As we have finite number of symbols (terminals and nonterminals) in V_N', then $W_k = W_{k+1}$ for some k. This means that $W_k = W_{k+j}$ for all $j \geq 0$.

Step 2. Construction of V_N'', Σ'' and P'. We define following :

$V_N'' = V_N' \cap W_k$

$\Sigma'' = \Sigma \cap W_k$

$P'' = \{ A \rightarrow \beta \mid A \in W_i \}$

Every symbol Z in G' appears in some sentential form, say $\alpha Z \beta$. Therefore $S \; \alpha Z \beta \; s$ for some , *i.e.*, G' is reduced.

ELIMINATION OF NULL AND UNIT PRODUCTIONS

- There are some straight forward ways to improve a context-free grammar without changing the resulting language.
- Certain types of productions are eliminated that may be difficult to handle and to work with them.
- The second step in the continuation is standardization of productions so that they all have a certain normal form.

Theorem 6.6 If $G = (V_N, \Sigma, P, S)$ is a context-free grammar, then we can find a context-free grammar G' having no null productions such that

$$L(G') = L(G) - \{\wedge\}.$$

PROOF. The difference between grammars G and G' is of only productions, so we define G' as (V_N, Σ, P, S). The construction steps are as follows :

Step 1. Construction of the set of nullable variables. We find nullable variables recursively. A variable E in a CFG is nullable if $E \Rightarrow \wedge$. Nullable variable are found as :

(i) $W_1 =$ is in $P\}$

(ii) $W_{i+1} = W_i \cup \{A' \in V_N \mid$ there exists a production $A' \rightarrow \beta$ with $\beta \in W'_i\}$

According to the definition of Wi, $Wi \subseteq Wi+1$ for all i, Wi is subset of $Wi+1$, which says that all element of Wi will definitely be in $Wi+1$ but it is not necessary that all elements of $Wi+1$ must be in Wi.

As V_N has finite number of variables therefore, $W_{k+1} = W_k$ for some $k \leq |V_N|$, where $|V_N|$ is the number of element in V_N. So $W_{k+j} = W_k$ for all j. Let $W = Wk$. where W is the set of all nullable variables.

Now we are going to remove all those productions that contain nullable variables.

Step 2. (i) Construction of P'. We will include all productions in P', which do not have any nullable variable in the right hand side of the production.

(ii) If $A' \rightarrow X_1X_2X_3....X_m$ is in P, the production of the $A' \rightarrow \beta_1\beta_2\beta_3... \beta_m$ are include in P', where $\beta_i = X_i$ if $X_i \notin W$. $\beta_i = X_i \cup \wedge$ if $X_i \in W$ and $\beta_1\beta_2\beta_3 ... \beta_m \neq \wedge$. Actually the production $A' \rightarrow X_1X_2X_3....X_m$ gives several productions in P'. The productions are obtained either by not removing any nullable variable on the right hand side of $A' \rightarrow X_1X_2X_3....X_m$ or by removing some or all nullable variable provided some symbol appeared on the right hand side after removing.

Note. In above theorem A' is not complement of A. We have assume A' as a random variable such that $A' \in V_N$.

Theorem 6.7 Let G be a context-free grammar with no null production, we can find a context-free grammar G' which has no unit production such that $L(G') = L(G)$.

PROOF. The productions of the form $D \rightarrow B$ are called unit productions, where, $D, B \in V_N$. Let A be any variable in V_N.

Step 1. First we construct the set of variable derivable form A. We define $Wi(A)$ recursively as follows :

$W_0(A) = \{A\}$

$W_{i+1}(A) = W_i(A) \cup \{B \in V_N \mid C \rightarrow B \text{ is in } P \text{ with } C \in W_i(A)\}$.

It means W_{i+1} is calculated in terms of W_i. $W_i(A)$ is subset of $W_{i+1}(A)$. As the number of variables in V_N are finite, then $W_{k+1}(A) = W_k(A)$ for some $k \leq |V_N|$ where $|V_N|$ tells the number of variables in V_N. Therefore, $Wk+1(A)$ will be equal to $W_k(A)$ for all $j \geq 0$. Here we introduce new set $W(A)$, defined as

$W(A) = W_k(A)$

where, $W(A)$ is the set of variable derivable from A.

Step 2. Construction of A-productions in G'. The A-productions in G' are of the form

$A \rightarrow \alpha$ whenever $B \rightarrow \alpha$ is in G with $B \in W(A)$ and $\alpha \notin V_N$

Now we define $G' = (V_N', \Sigma, P', S)$ where P' is constructed using step 2 for every $A \in V_N$ and V_N' is set of variables in P'

Theorem 6.8 There exists an algorithm to decide whether $\wedge \in L(G)$ for a given context-free grammar G.

PROOF. $\wedge$ is in $L(G)$ if and only if S (the start symbol) is nullable. The construction given by theorem 6.7 terminates in a finite number of steps. Therefore we can prove this theorem by using two following steps :

Step 1. First we construct set of nullable variable (say W).

Step 2. In this step we test whether $S \in W$.

Hence, a context-free grammar that does not generate the string $\wedge$ can be written without $\wedge$-productions.

Chomsky Normal Form (CNF)

In Chomsky normal form, we restrict the length of V_N and the nature of symbols in the right hand side of productions. A context-free grammar G is said to be in Chomsky normal form if every production is of the form either $A \rightarrow a$, (exactly one terminal in the right hand side of the production) or $A \rightarrow BC$ (exactly two variables in the right hand side of the production).

For example, the context-free grammar G with productions $S \rightarrow AB$, $A \rightarrow a$, $B \rightarrow b$ is in Chomsky normal form.

$S \rightarrow \wedge$ is also in CNF if $S \rightarrow \wedge$ is in G such that $\wedge \in L(G)$ and S does not appear in the right hand side of any production. In all other cases it is must to eliminate null productions before reducing it into CNF.

Note. ***Several text book authors do not allow*** $S \rightarrow \wedge$ ***in Chomsky normal form.***

Reduction to CNF

Let us consider a context free grammar having productions $S \rightarrow AB \mid aC$, $A \rightarrow b$, $B \rightarrow aA$ and $C \rightarrow ABS$. Except $S \rightarrow aC$, $B \rightarrow aA$, $C \rightarrow ABS$, all the other productions are in the form required to be in CNF. For the production of the form $S \rightarrow aC$, we introduce new variable C_a such that $S \rightarrow aC$ can be replaced by

$$S \rightarrow C_aC,\ C_a \rightarrow a$$

which has the same derivation as $S \rightarrow aC$ has, but the productions $S \rightarrow C_aC$, $C_a \rightarrow a$ are in the form required for CNF Similarly $B \rightarrow aA$ can be replaced by

$$B \rightarrow C_aA$$

and there is no need to write $C_a \rightarrow a$ again.

For the production of the form $C \rightarrow ABS$ we introduce another new variable (say D) such that $C \rightarrow ABS$ can be replaced by

$$C \rightarrow AD,\ D \rightarrow BS$$

which are in required form. For long work strings, some more new variables can be introduced to put the production into CNF.

Context-free grammar in Chomsky normal form are very useful to determine whether or not it can generate any word at all. This is the decidability concept of emptiness in CFGs. Also, Chomsky normal form is particularly useful for programs that have to manipulate grammars.

Theorem 6.9 For every context-free grammar there is an equivalent grammar in Chomsky Normal Form (CNF).

PROOF. Let $G = (V_N, \Sigma, P, S)$ be a CFG. A context free grammar G is converted in to Chomsky Normal Form (CNF) by using following steps:

Step 1. Eliminate null and unit production by applying *theorem* 6.6 and *theorem* 6.7 respectively. Let the modified grammar thus obtained be

$$G_1 = (V_N', \Sigma', P', S)$$

Step 2. Elimination of terminals from right-hand side. We define grammar $G_2 = (V_N'', \Sigma', P'', S)$ where P'' and V_N'' are constructed as follows :

All the productions in P' of the form $A \to b$ or $B \to CD$ are included in P'', and all the variables in V_N' are included in V_N''.

If there is a production of the form $X \to a_i A_1 A_2 A_3 \ldots\ldots A_n$ with some terminals on right hand side, say A_i is terminal with value a_i, add a new variable C_{a_i} to V_N'' and add a production $C_{a_i} \to a_i$ to P''.

Every terminal in right hand side of the production $X \to A1A2A3\ldots\ldots An$ is replaced by the corresponding new variable and the variables on right hand side are retained. The resulting production is included to P''. This way we get $G_2 = (V_N'', \Sigma', P'', S)$.

Step 3. Restricting the number of variables on right hand side. Every production in P'' must contain either a single terminal (or $\wedge$ in case of $S \to \wedge$) or two more variables in the right hand side.

We define grammar in CNF $G' = (V_N''', \Sigma', P''', S)$ as follows :

(i) All productions in P'' are added to P''' if they are in required form and, all the variables in V_N'' are added to V_N'''.

(ii) If there are productions of the form $A \to A1A2A3\ldots\ldots An$, where $n \geq 3$, we introduction new productions $A \to A1C1$, $C1 \to A2C2$, $C2 \to A3C2$, $C3 \to C3\ A4$, $C4 \to \ldots\ldots$, $Cn\text{–}2 \to An\text{–}1An$ and new variables $C1, C2, C3, \ldots Cn\text{–}2$. These productions are added to P''' and new variables introduced in these productions (*e.g.*, $C1, C2, \ldots\ldots Cn\text{–}2$) are added to V_N'''. Thus, the grammar $G' = (V_N''', \Sigma', P''', S)$ is in CNF.

Greibach Normal Form (GNF)

A context-free grammar is said to be in Greibach normal form if every production is of the form

$$A \rightarrow a\alpha$$

where $a \in \Sigma$, $A \in V_N$ and $\alpha \in V_N^*$. It shows that α is a string of variables having zero or more length.

- **Grammars in Greibach normal form are typically complex and much longer than the context-free grammars from which they were derived.**
- **Greibach normal form is useful for proving the equivalence of context-free grammar (CFGs) and Nondeterministic Push Down Automata (NPDA).**
- **When we convert a CFG into an NPDA, or vice versa, we use GNF.**

Lemma 6.1 If $G = (V_N, \Sigma, P, S)$ is a given context-free grammar, and an A-production $A \to B\gamma$ is in P. If B-productions are

$B \to \beta_1 \mid \beta_2 \mid \beta_3 \mid \beta_4 \mid \beta_5 \mid \beta_6 \ldots\ldots\ldots \beta_k$.

If we define a new set of productions P_1 as

$P_1 = (P - \{ A \to B\gamma \}) \cup \{ A \to B_i \gamma \mid 1 \le i \le k \}$

(*i.e.*, productions of the form $A \to B\gamma$ are eliminated by introducing the productions of the form $A \to B_i \gamma$). Then grammar $G_1 = (V_N, \Sigma, P_1, S)$ is equivalent to given context-free grammar G.

PROOF. If we apply $A \to B\gamma$ in some derivation $s \in L(G)$, we have to eliminate B from grammar G which is the same as applying $A \to B_i \gamma$ for some i in grammar G_1.

Hence, $s \in L(G_1)$, *i.e.*, $L(G_1) \subseteq L(G)$

Similarly, instead of applying $A \to B_i \gamma$, we can also apply $A \to B\gamma$ and $B \to \beta_i$ to get

$A \quad B_i \gamma \quad \overset{*}{\underset{G}{\Rightarrow}}$

This shows that, $L(G_1) \subseteq L(G)$

This way, we can delete a variable B appearing as the first symbol on the right side of some A-productions. It provides no B-production which has B as the first symbol on right side.

In the construction, given by *Lemma* 6.1, B is eliminated from the production $A \to B\gamma$ be simply replacing B by the right side of every B-production.

Therefore the *Lemma* 6.1 is useful to eliminate A from the right side of $B \to AB$.

Lemma 6.2 If $G = (V_N, \Sigma, P, S)$ is a context-free grammar, and A-productions are

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid A\alpha_4 \ldots\ldots\ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots\ldots\ldots \beta_n$$

where β_i's do not start with A. Suppose Z is a new variable ($Z \notin VN$). Suppose a context-free grammar G_1 is defined as

$$G_1 = (\{V_N \cup Z\}, \Sigma, P_1, S)$$

where P_1 is defined as follows :

(i) The set of A-productions in P_1 are

$$A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots\ldots\ldots \beta_n \text{ and } A \rightarrow \beta_1 Z \mid \beta_2 Z \mid \beta_3 Z \mid \ldots\ldots\ldots \beta_n Z.$$

(ii) The set of Z-productions in P_1 are

$$Z \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \alpha_4 \ldots\ldots\ldots \mid \alpha_m \text{ and } Z \rightarrow \alpha_1 Z \mid \alpha_2 Z \mid \alpha_3 Z \mid \alpha_4 Z \ldots\ldots\ldots \mid \alpha_m Z$$

(iii) The productions for the other variables are the same as in P.

Then, context-free grammar G_1 is equivalent to G.

PROOF. Let us consider a leftmost derivation $s \in L(G)$ for the proof of $L(G) \subseteq L(G_1)$. The only productions in $P - P_1$ are

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid A\alpha_4 \ldots\ldots\ldots \mid A\alpha_m$

If $A \rightarrow A\alpha_{i_1} \mid A\alpha_{i_2} \mid \ldots\ldots \mid A\alpha_{i_k}$ are used, then $A \rightarrow \beta_j$ should be used at a later stage to eliminate A. So we have derivation

$$A \underset{G}{\Rightarrow}{}^{*} \beta_j A\alpha_{i_1}\alpha_{i_2}\ldots\ldots\alpha_{i_k}$$

While driving $s \in L(G)$. However, by using CFG G_1 we have

$$A \underset{G_1}{\Rightarrow} \beta_j Z \underset{G_1}{\Rightarrow} \beta_j \alpha_{i_1} Z \underset{G_1}{\Rightarrow} \beta_j \alpha_{i_1} Z\alpha_{i_2} Z$$

$$\underset{G_1}{\Rightarrow}{}^{*} \beta_j \alpha_{i_1}\alpha_{i_2}\alpha_{i_3}\ldots\ldots\ldots\alpha_{i_k}$$

i.e., $A \underset{G_1}{\Rightarrow}{}^{*} \beta_j A\alpha_{i_1}\alpha_{i_2}\alpha_{i_3}\ldots\ldots\ldots\alpha_{i_k}$

This way, 'A' can be eliminated by using the productions in grammar G_1. Therefore, $s \in L(G_1)$.

For the proof of $L(G_1) \subseteq L(G)$, let us consider a leftmost derivation of $s \in L(G_1)$.

The only productions in $P_1 - P$ are

$A \rightarrow \beta_1 Z \mid \beta_2 Z \mid \beta_3 Z \mid \ldots\ldots\ldots \beta_n Z$. and $Z \rightarrow \alpha_1 Z \mid \alpha_2 Z \mid \alpha_3 Z \mid \alpha_4 Z \ldots\ldots\ldots \mid \alpha_m Z$

If the new introduced variable Z appears in the derivation of s, is due to the application of the production $A \rightarrow \beta_j Z$ in some starting step. Also, Z can be eliminated only by a production of the form $Z \rightarrow_* \alpha_i \mid \alpha_j Z$ for some i and j in later steps. Therefore, we get

$$A \underset{G_1}{\Rightarrow} \beta_j \alpha_{i_1} \alpha_{i_2} \alpha_{i_3} \ldots\ldots\ldots \alpha_{i_k}$$

in the derivation of s. Also, we know that $\beta_j \alpha_{i_1} \alpha_{i_2} \alpha_{i_3} \ldots\ldots\ldots \alpha_{i_k} \underset{G_1}{\overset{*}{\Rightarrow}} A$. Therefore, $s \in L(G)$, which shows that $L(G_1) \subseteq L(G)$.

Theorem 6.10 Every context-free language L can be generated by a context-free grammar G in Greibach normal form. (*i.e.*, a context-free grammar can be reduced to Greibach normal form).

PROOF. We prove the theorem when $\wedge$ is not in L (*i.e.*, $\wedge \notin L$) and then we extend the construction to $\wedge \in L$. Thus there are two cases first when $\wedge \notin L$ and second when $\wedge \in L$.

Case 1. Construction of G (when $\wedge \notin L$).

Step 1. We eliminate null productions and then construct a grammar G in Chomsky Normal form which generates language L. We rename the variables as $A_1, A_2, A_3, \ldots\ldots, A_n$ with $S = A_1$. Therefore, grammar G is defined as

$G = (V_N, \Sigma, P, S)$

$V_N = \{A_1, A_2, A_3, \ldots\ldots, A_n\}$

with $S = A1$

Step 2. To obtain the productions in the form $A_i \rightarrow a\gamma$ or $A_i \rightarrow a_j\gamma$, where $i < j$, we convert the A_i-productions (for $i = 1, 2, 3, \ldots, n-1$) in the form $A_i \rightarrow a_j\gamma$, such that $i < j$. The proof of such modification can be done by induction on i.

Let us consider $A1$-productions. If some $A1$-productions are of the form $A1 \rightarrow A1\gamma$, then we apply *Lemma* 6.2 to eliminate such productions. By introducing a new variable (say $Z1$), we get $A1$-productions in the form $A1 \rightarrow a$ or $A1 \rightarrow A_j\gamma'$, where j is greater than 1, which is the basis for induction.

This way we can modify A_1-productions, A_2-productions, A_3-productions, ..., A_i- productions. Let us consider A_{i+1} productions. There is no need of any modification in productions of the form $A_{i+1} \rightarrow a\gamma$. Now we consider the first symbol in the right side of remaining A_{i+1}-productions. Suppose k is the smallest index among the indices of such variables.

If $k > i+1$, then there is no need to prove anything. Otherwise we apply induction hypothesis to A_k-productions for $k \leq i$. Therefore, any A_k-production is of the form $A_k \rightarrow A_j\gamma$ where $j > k$ or $A_k \rightarrow a\gamma'$. Now we can apply *Lemma* 6.1 to $Ai+1$-productions whose right side starts with A_k, then the modified $Ai+1$ productions are of the form

$A_{i+1} \rightarrow A_j\gamma$

where $j > k$ or $A_{i+1} \rightarrow a\gamma'$.

We repeat the above construction by finding k for the new set of $Ai+1$-productions. Finally, the $Ai+1$-production are converted to the form $Ai+1 \rightarrow A_j\gamma$, where $j \geq i+1$ or $Ai+1 \rightarrow a\gamma'$. The productions of the form $Ai+1 \rightarrow Ai+1\gamma$ can be modified by applying *Lemma* 6.2.

This way, we have converted $Ai+1$-productions in the required form. Any Ai-production is of the $Ai \rightarrow A_j\gamma$, where $i < j$ or $Ai \rightarrow a\gamma'$ and An-production is of the form $An \rightarrow An\ \gamma$ or $An \rightarrow a\gamma'$ at the end of this step.

Step 3. We convert A_n-production to the form $A_n \rightarrow a\gamma$. The productions obtained in step 2 of the form $A_n \rightarrow A_n\gamma$ are eliminated by applying *Lemma* 6.2. As a result the A_n-production are of the form $A_n \rightarrow a\gamma$.

Step 4. Now we modify A_i-productions to the form $A_i \rightarrow a\gamma$ (for $i = 1, 2, 3, \ldots, n-1$). At the end of step 3, the A_n-productions are of the form $A_n \rightarrow a\gamma$, the A_{n-1} productions are of the form $A_{n-1} \rightarrow a\gamma'$ or $A_{n-1} \rightarrow A_n\gamma$. We eliminate the productions of the form $A_{n-1} \rightarrow A_n\gamma$ by applying *Lemma* 6.1. The resulting A_{n-1}-productions are in the required form. We repeat the similar constructions for A_{n-2}, A_{n-3},..... productions.

Step 5. In this step we modify Z_i-productions. We apply *Lemma* 6.2 every time, and we get a new variable. As a result Z_i -productions are of the form $Z_i \rightarrow \alpha Z_i$ or $Z_i \rightarrow \alpha$, where α is obtained from $A_i \rightarrow A_j\alpha$. Thus the Z_i-productions are of the form $Z_i \rightarrow a\gamma$ or $Z_i \rightarrow A_k\gamma$ for some k. At the end of step 4, the right side of any A_k-production was starting with a terminal, which we are going to change in this step. Therefore we can apply *Lemma* 6.1 to eliminate the productions of the form $Z_i \rightarrow A_k\gamma$. Thus, the resultant grammar G_1 obtained at the end of this step (step 5) is in GNF.

Case 2. Construction of G (when $\wedge \in L$. We repeat the construction as in case 1, to get a grammar $G' = (V_N', \Sigma, P', S)$ in Greibach normal form such that $L(G') = L - \{\wedge\}$. We define a new grammar G_1 as

$$G_1 = (V_N' \cup \{S\}, \Sigma, P' \cup \{S' \rightarrow S \mid \wedge\}, S)$$

The production $S' \rightarrow S$ can be eliminated by using theorem to eliminate the unit productions. Thus all S- and S' productions are in required form and G_1 is in Greibach normal form.

Thus we cab apply steps 2 to 5 to a context-free grammar, whose productions are of the form $A \rightarrow A\alpha$ or $A \rightarrow \alpha$, where $\alpha \in$ and $|\alpha| \geq 2$ (the length of α is 2 or more), to reduce it into GNF, by first converting the productions of the form $A \rightarrow \alpha$ (where $\alpha \in$ and $|\alpha| \geq 2$) into Chomsky normal form (CNF).

CYK ALGORITHM

In this section we will see that the parsing of a string 's' requires $|s|^3$ steps. We now discuss CYK algorithm to have this claim.

The name CYK is taken from Cocke J., Younger D. H., and Kasami T., because of their contribution to this concept.

The major limitation of this algorithm is that the grammar input must be in Chomsky Normal Form (CNF).

Let us assume a CFG

$G = (V_N, \Sigma, P, S)$ in CNF, and

there is a string , $s = x_1x_2x_3x_4.....x_n$.

Let us define substrings , $s_{ij} = x_ix_{i+1}.....x_j$

and a subset of nonterminals V_{ij} in V_N, as

$$V_{ij} = \{ X \in V \mid X \Rightarrow^{*} s_{ij} \}$$

This means $s \in L(G)$ iff $S \in V_{1n}$.

For the computation of Vij , we observe that $X \in V_{ii}$ iff CFG G contains a production of the form $X \rightarrow x_i$. this way we can compute V_{ii} for all i from 1 to n using productions of the grammar G. For $i > j$, we see that X drives s_{ij} iff there exists a production of the form $X \rightarrow YZ$ with Y s_{ik} and Z s_{k+1} for some k varying between i and j. This can also be represented as

$$V_{ij} = \bigcup_{k \in \{i, i+1, \ldots, j\}} \{ X \mid X \rightarrow YZ \text{ and } Y \in V_{ik}, Z \in V_{k+1,j} \}$$

This equation shows the flexibility for computation of all Vij. We can compute

(i) $V_{11}, V_{22}, V_{33}, \ldots\ldots, V_{nn}$

(ii) $V_{12}, V_{23}, V_{34}, \ldots\ldots, V_{n-1, n}$

(iii) $V_{13}, V_{24}, V_{35}, \ldots\ldots, V_{n-n, n}$

With the help of CYK algorithm we can determine the membership for any given language for which there exists a CFG in CNF. We can see that algorithm requires $O(n^3)$ steps as there are sets of V_{ij}. The evaluation of each set involves n terms.

Example 6.15 Determine whether the string $s = 00111$ belongs the language generated by the following CGF in Chomsky Normal Form:

$S \rightarrow XY, X \rightarrow 0 \mid YY, Y \rightarrow 1 \mid XY$

Solution. We have $s_{11} = 0$, the term V_{11} can be seen as set of all nonterminals that derives 0, therefore $V_{11} = \{X\}$. Now $s_{22} = 0$, so we have $V_{22} = \{X\}$, and this way we have

$V_{11} = \{X\}, V_{22} = \{X\}, V_{33} = \{Y\}, V_{44} = \{Y\}, V_{55} = \{Y\}$

By using formula , we have

$V_{12} = \{X \mid X \rightarrow YZ, Y \in V_{11}, Z \in V_{22}\}$

As we have $V_{11} = \{X\}$ and $V_{22} = \{X\}$, therefore the set will contain all nonterminals that occur on the left hand side of the production whose right hand side is XX. Since there is no such production. Thus V_{12} is empty. In this continuation we have,

$V_{23} = \{X \mid X \rightarrow YZ, Y \in V_{22}, Z \in V_{33}\}$

Therefore the required right hand side is XY, so we have

$V_{23} = \{S, Y\}$.

In the same way we can compute $V_{34} = \{X\}$, $V_{45} = \{X\}$, and

$V_{13} = \{S, Y\}, V_{24} = \{X\}, V_{35} = \{S, Y\}, V_{14} = \{X\}, V_{25} = \{S, Y\}, V_{15} = \{S, Y\}$

The leveled computation is represented by following diagram:

Level 5	*S, Y*				
Level 4	*X*	*S, Y*			
Level 3	*S, Y*	*X*	*S, Y*		
Level 2		*S, Y*	*X*	*X*	
Level 1	*X*	*X*	*Y*	*Y*	*Y*
Input String	0	0	1	1	1

Therefore, 00111 ∈ *s*.

APPLICATIONS OF CFG

Context free grammar is a useful tool for defining programming languages.

Here, we will discuss how context free grammar is used to define the syntax of C language and definition part of HTML (Hyper Text Markup Language).

By using CFG and BNF, here are some C statements :

<statement> ::= <assignment statement> | < compound-statement> | <iteration-statement > |<selection-statement >

The productions in BNF for iteration statements are:

<iteration-statement> ::= do < statement >while (<expression>) |
while(<logical-expression>) <statement>|
for ([<expression>]; [<expression>]; [<expression>];) <statement>

The productions in BNF for selection-statements are:

<selection-statement > ::= if (<logical-expression >)<statement>|
if (<logical-expression >)<statement> else <statement> |
switch(<expression>) {<cases>}

The productions in BNF for logical-expression are:

<logical-expression> ::= <comparison>|<comparison>&&<logical-expression>|
<comparison>||<logical-expression>

The productions in BNF for comparison are:

<comparison> ::= (<boolean-operand> <comparison-operator>(<boolean-operand>)

The productions in BNF for boolean-operand are:

<boolean-operand> ::= True |False |<identifier>

The productions in BNF for expression are:

<expression> ::= <factor> |<expression> + <factor>| <expression> -<factor>

The productions in BNF for factor are:

<factor> ::= <factor>*<operand> | <operand>

The productions in BNF for comparison-operator are:

<comparison-operator> ::= > | < | >= |<= |!= | =

The productions in BNF for cases are:

<cases> ::= <case><cases> | <default>

The productions in BNF for case are:

<case> ::= <case-head> <cases> |<case-head> <statement>

<case-head> ::= case<literal> :

The production in BNF for default case is:

<default> ::= default : <statement-sequence>

Let us now define some token classes in terms of BNF notation.

<input> ::= <token> | <comment> | <white-space>

<white-space> ::=\n |\t | \r | <invisible-space>

<comment> ::= //<sequence of characters> /* <sequence of characters> */

<token> ::= <identifier> |<operator> |<literal> |<keyword> |<separator>

<identifier> ::= <letter>(<letter> |<digit>)*

<letter> ::= *a|b|c|d|.....x|y|z|A|B|C|D|.....|X|Y|Z*

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<keyword> ::= int |main| void|getch| if| else | boolean

<literal> ::= <integer-value> |<float-value> |<boolean-value>

<integer-value> ::= (digit)(digit)*

<boolean-value> ::= True | False

<float-value> ::= <decimal> |<exponential>

<decimal> ::= <signed-integer> .<integer>|<integer-value>|<sign>